.NET Full Stack Development Program

—

Day 11 ADO.NET

Outline

- Introduction to ADO.NET
- Database handling via ADO.NET – DataReader
- Another Data Fetching Approach – DataAdapter
- Transaction Management in ADO.NET

Introduction to ADO.NET

ADO.NET

- ADO.NET(ActiveX Data Object.NET) is a data access technology from the Microsoft .NET Framework that provides communication between our application and database.

- ADO.NET provides consistent access to data sources such as SQL Server and XML, and to data sources exposed through OLE DB and ODBC. Data-sharing consumer applications can use ADO.NET to connect to these data sources and ***retrieve***, ***handle***, and ***update*** the data they contain.

- Now we are only focusing on RDBMS (Most .NET positions are related to RDBMS, especially in MS SQL Server, non-relational DB is a plus)

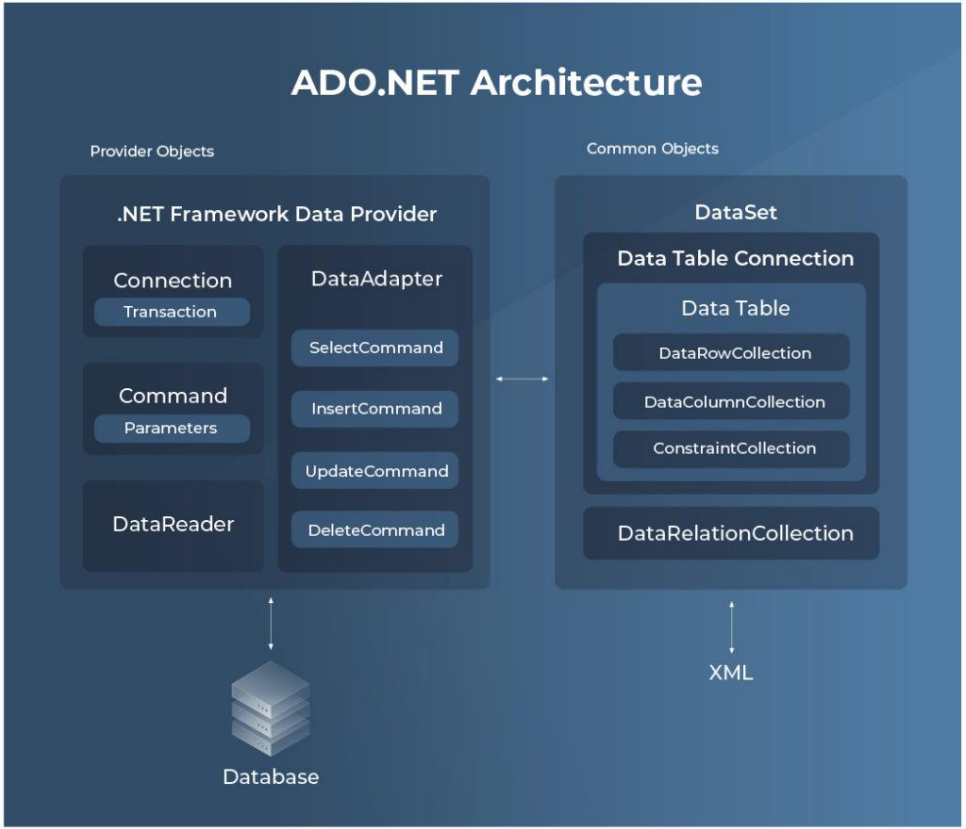
ADO.NET Components

The two main components of ADO.NET for accessing and manipulating data are

- .NET Framework data providers
  - The **.NET Framework Data Providers** are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data.
  - Ex. Connection, Command, DataReader, DataAdapter

- DataSet
  - The ADO.NET **DataSet** is explicitly designed for data access **independent of any data source**. As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application.

ADO.NET Architecture



Database handling via ADO.NET

Database Handling Via ADO.NET

1) Open Connection
2) Create Command
3) Execute command and obtain result
4) Iterate through the results (DataReader) / directly obtain results (DataAdapter)
5) Close connection

1) Establish Connection to SQL Server

```
//Establish Connection
SqlConnection conn = new SqlConnection(connString);
//Open Connection
conn.Open();
//Do something here

//Close Connection
conn.Close();
```

You can also

- Connect to an OLE DB Data Source via OleDbConnection
- Connect to an ODBC Data Source via OdbcConnection
- Connect to an Oracle Data Source via OracleConnection

Connection String

Sql Server Authentication Syntax:

"Persist Security Info=False;User ID=*****;Password=*****;Initial Catalog=DatabaseName;Server=ServerName”

*Define within config file

*We can use other syntax: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/connection-string-syntax

Connection Pooling

Connecting to a data source can be expensive and time consuming. To minimize the cost of opening connections, ADO.NET uses an optimization technique called *connection pooling*, which minimizes the cost of repeatedly opening and closing connections.

- **SQL Server Connection Pooling (ADO.NET)**
  - Default (whether to use connection pool)
  - Pooling=true
  - Max Pool Size
  - Min Pool Size

Option 1: try catch finally

```
public static readonly string connString =
"Persist Security Info=False;User ID=SA;Password=Beaconfire1234;Initial Catalog=Student;Server=localhost";
0 references
static void Main(string[] args)
{
    SqlConnection conn = null;
    try
    {
        conn = new SqlConnection(connString);
        conn.Open();
        Console.WriteLine(conn.Database);
    }
    catch(Exception ex)
    {
        Console.WriteLine("Something wrong with the databse connection: " + ex.Message);
        throw ex;
    }
    finally
    {
        conn.Close();
    }
}
```

Option 2: using
(Preferred way)

```
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    Console.WriteLine(conn.Database);
    // we don't need to explicitly close the connection
    // however if you want to catch exceptions, you can surround your using block with try catch blocks
}
```

2) Create a Command

SqlCommand with no parameter

```
string query = "SELECT ClassName FROM StudentClass WHERE ClassId = 1";
#region Without Parameters
using (SqlConnection conn = new SqlConnection(connString))
{
    //Establish Connection
    conn.Open();
    //Create the SqlCommand
    SqlCommand cmd = new SqlCommand(query, conn);
    //Use the command do something here
}
#endregion
```

Command

Some important Properties in SqlCommand class:

- Connection
  - Specify which connection to use
- CommandText
  - Sql that command processed
- CommandType(enum)
  - **CommandType.Text** – meaning it's going to execute a sql query
  - **CommandType.StoredProcedure** – meaning it's going to execute a stored procedure
- Parameters
  - It is used to add the input parameter
- Transaction
  - Success all or fail all

Parameterized Command

```
string parameterizedQuery = "SELECT ClassName FROM StudentClass WHERE ClassId = @ClassId";
#region With Parameters
using (SqlConnection conn = new SqlConnection(connString))
{
    //Establish Connection
    conn.Open();
    //Create the SqlCommand and set its properties
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = conn;
    cmd.CommandText = parameterizedQuery;
    cmd.CommandType = CommandType.Text; // the default CommandType is Text, so this line is optional;

    //get the parameter
    int resultClassId = 1;

    //Add the input parameter and set its properties
    SqlParameter parameter = new SqlParameter();
    parameter.ParameterName = "@ClassId";
    parameter.SqlDbType = SqlDbType.Int;
    parameter.Value = resultClassId;
    //Add the parameter to the Parameters collection
    cmd.Parameters.Add(parameter);
    //Use the command do something here
}
#endregion
```

3)Execute Command

- ExecuteScalar
  - used to execute SQL Command or stored procedure, after executing, return **a single value(object)** from the database
  - returns the first column of the first row in the result set from a database, additional columns or rows are ignored.
- ExecuteNonQuery
  - used to execute SQL Command or the stored procedure performs INSERT, UPDATE, or DELETE operations.
  - it returns an integer specifying the number of rows inserted, updated or deleted.
  - doesn't return any data from the database
- ExecuteReader
  - used to execute a SQL Command or stored procedure returns a set of rows from the database and stores the results in DataRader.

SqlDataReader

- The **DataReader** provides an **unbuffered stream of data** that allows procedural logic to efficiently process results from a data source sequentially. The DataReader is a good choice when you're retrieving large amounts of data because the data is not cached in memory.

- The **DataReader** is **read-only**, it’s not possible to change the data using DataReader.

- Use the **DataReader.Read()** method to **obtain a row from the query results.**

- **Always** call the **Close()** method when you have finished using the DataReader object

- While a DataReader is **open**, the **Connection is in use exclusively by that DataReader**. You cannot execute any commands for the Connection, including creating another DataReader, until the original DataReader is closed.

4) Iterate through the results (DataReader)

```
//SqlDataReader Example
string query3 = "SELECT * FROM StudentClass";
using (SqlConnection conn = new SqlConnection(connString))
  {
     //Establish Connection
     conn.Open();
    //Create the SqlCommand
     SqlCommand cmd3 = new SqlCommand(query3, conn);
    //Execute the command

     using (SqlDataReader reader = cmd3.ExecuteReader())
    {
        //We can present the data use a while loop
        //or hold all the data in a collection for future use.

         //while (reader.Read())
        //{
        //   Console.WriteLine(reader["ClassId"] + ", " + reader["ClassName"]);
        //}

        //Store the result in a List
         List<StudentClass> studentClasses = new List<StudentClass>();
         while (reader.Read())
           {
             studentClasses.Add(
               new StudentClass()
                 {
                   ClassId = Convert.ToInt32(reader[0]),
                   ClassName = reader[1].ToString()
                 });
           };
           //Present the List
           foreach (var classInfo in studentClasses)
             {
               Console.WriteLine("Class Id: " + classInfo.ClassId +
                "  Class Name: " + classInfo.ClassName);
             };
      }
    }
```

```
Class Id: 1  Class Name: SQL Batch
Class Id: 2  Class Name: C# Batch
Class Id: 3  Class Name: Java Batch
Class Id: 4  Class Name: Data Batch
Class Id: 5  Class Name: .Net Batch
```

With Parameter

```
1; SQL Batch
```

```
string parameterizedQuery = "SELECT * FROM StudentClass WHERE ClassId = @ClassId";
#region With Parameters
using (SqlConnection conn = new SqlConnection(connString))
{
    //Establish Connection
    conn.Open();
    //Create the SqlCommand and set its properties
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = conn;
    cmd.CommandText = parameterizedQuery;
    cmd.CommandType = CommandType.Text; // the default CommandType is Text, so this line is optional;

    //get the parameter
    int resultClassId = 1;

    //Add the input parameter and set its properties
    SqlParameter parameter = new SqlParameter();
    parameter.ParameterName = "@ClassId";
    parameter.SqlDbType = SqlDbType.Int;
    parameter.Value = resultClassId;
    //Add the parameter to the Parameters collection
    cmd.Parameters.Add(parameter);

    //Execute the command
    using (SqlDataReader reader = cmd.ExecuteReader())
    {
        if (reader.HasRows)
        {
            while(reader.Read())
            {
                Console.WriteLine("{0}; {1}", reader[0], reader[1]);
            }
        }
    }
}
#endregion
```

5) Close Connection

Just like file I/O, we have to close the database connection after we finish all the jobs

- reader.Close() – if don't use using()
- conn.Close() – if don't use using()

DataAdapter Approach

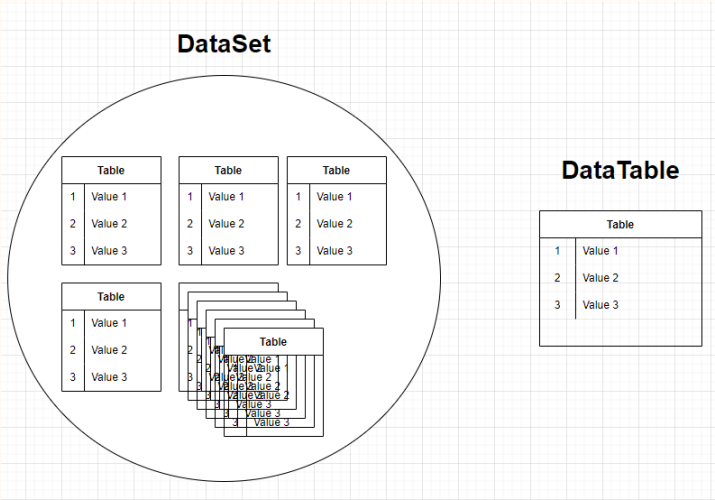
DataAdapter

A **DataAdapter** is used to retrieve data from a data source and populate tables within a **DataSet**. The DataAdapter also resolves changes made to the DataSet back to the data source. The DataAdapter uses the **Connection** object of the .NET Framework data provider to connect to a data source, and it uses **Command** objects to retrieve data from and resolve changes to the data source.

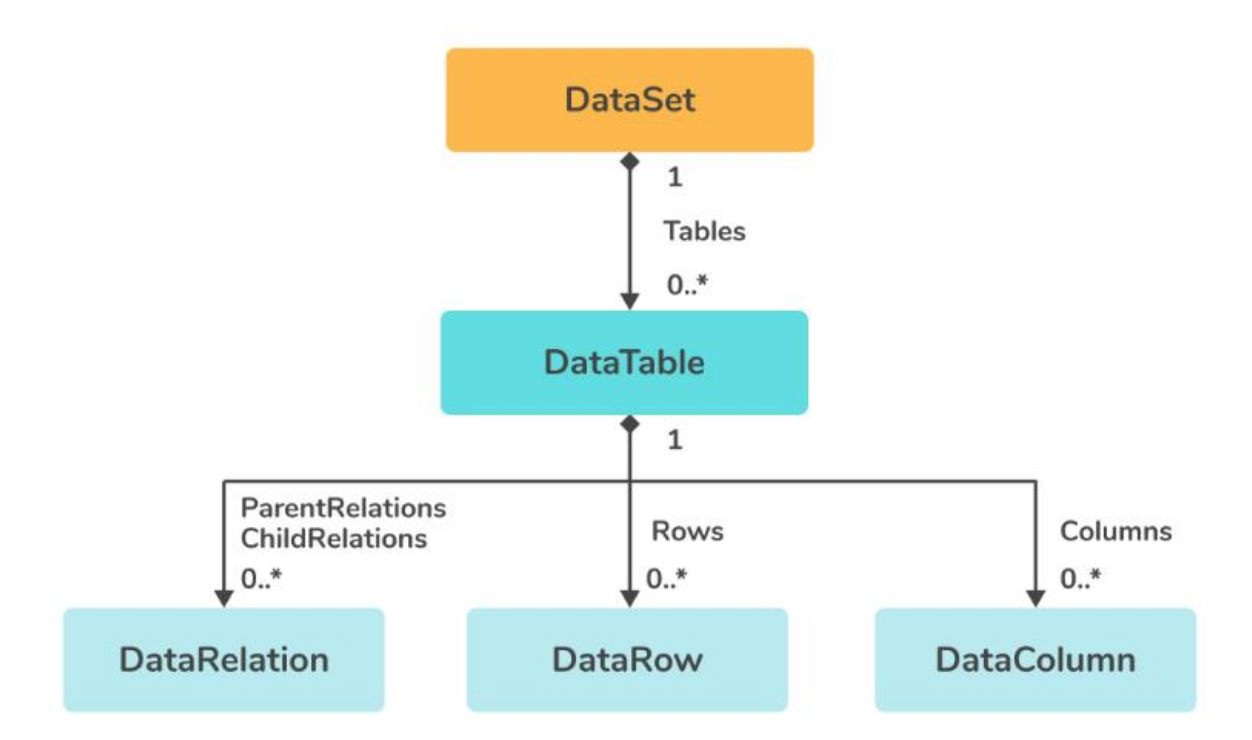
```
public static readonly string connString =
"Persist Security Info=False;User ID=SA;Password=Beaconfire1234;Initial Catalog=Student;Server=localhost";
0 references
static void Main(string[] args)
{
    string query = "SELECT * FROM StudentClass";
    using (SqlConnection conn = new SqlConnection(connString))
    {
        //Establish connection
        conn.Open();
        //Create an instance of the SqlDataAdapter
        SqlDataAdapter adapter = new SqlDataAdapter(query, conn);
        //Do something else afterwards...
    }
```

DataTable & DataSet

- **DataTable & DataSet**
  - DataTable represents one table that contains the relational data in memory
  - Dataset is simply the collection of data tables
  - Normally, those two are used along with **DataAdapter** for fetching data from database



DataSet Structure



DataTable

```
#region Create a Data Table Example
DataTable studentTable = new DataTable("Student");
DataColumn stuId = new DataColumn();
stuId.ColumnName = "StuID";
stuId.DataType = typeof(int);
studentTable.Columns.Add(stuId);

studentTable.Columns.Add("StuName", typeof(string));
studentTable.Columns[1].AllowDBNull = false; // student name cannot be null

//Set primary key
studentTable.PrimaryKey = new DataColumn[] { stuId }; // can also use index: studentTable.Columns[0]
//Set unique constraints
studentTable.Constraints.Add(new UniqueConstraint(studentTable.Columns[1]));
#endregion
```

StuId	StuName

DataTable

DataTable represents one table of in-memory relational data.

Common methods used in DataTable:

- **AcceptChanges()** – commit changes
- RejectChanges() – rollback changes
- Clear() – clear data
- Copy() – copy the schema as well as data
- Clone() – copy the schema only but not data
- Load(IDataReader) – load DataReader to a DataTable
- Merge(DataTable) – combine two DataTables
- NewRow() – create a DataRow
- Reset() – reset DataTable to the original status
- Select() – fetch an array of DataRow including all data with condition and sorting

DataRowState

- Detached
  - The row has been created but is not part of any DataRowCollection. A DataRow is in this state immediately after it has been created and before it is added to a collection, or if it has been removed from a collection
- Unchanged
  - The row has been **modified** and **AcceptChanges()** has not been called
- Added
  - The row has been **added** to a DataRowCollection, and **AcceptChanges()** has been called
- Deleted
  - The row was deleted using the **Delete()** method of the DataRow
- Modified
  - The row has **not changed** since **AcceptChanges()** was last called

DataSet

Dataset is simply the collection of datatables

*It loads data into memory which is **fast** and **efficient**

*It's independent from the original resource

How is DataSet being used?

- Use DataAdapter to populate database data into DataSet
- Use DataAdapter to commit DataSet data to database
- Load XML file to DataSet

DataSet

Common methods used in DataSet:

- AcceptChanges() – commit changes
- RejectChanges() – rollback changes
- Clear() – clear data
- Copy() – copy the schema as well as data
- Clone() – copy the schema only but not data
- Load(IDataReader) – load DataReader to a DataSet
- Merge(DataTable/DataSet) – combine
- Reset() – reset DataSet to the original status

DataSet Relations

dt ⇒ StudentTable

studentId	studentName

dt2 ⇒ ScoreTable

scoreId	score	studentId

- One-to-One
- One-to-Many
  - *ForeignKeyConstraints is added to the many side
  - *parentColumn refers to the one side
- Many-to-Many

```
DataSet ds = new DataSet();
ds.DataSetName = "ds1";
ds.Tables.Add(dt);
//ds.Relations.Add();
DataTable dt2 = new DataTable("Score");
dt2.Columns.Add("scoreId", typeof(int));
dt2.Columns.Add("score", typeof(double));
dt2.Columns.Add("studentId", typeof(int));
DataRelation dataRelation = new DataRelation("studentScoreRelations",  dt.Columns[0], dt2.Columns[2], true);
//true --- means it will generate the foreignKeyConstraints automatically
ds.Tables.Add(dt2);
ds.Relations.Add(dataRelation);
```

Example of One-to-Many

DataSet Relations

- Read data based on Relation

```
//read child(many) from parent(one)
DataRow[] rows = dt.Rows[0].GetChildRows(dataRelation);
foreach (DataRow row in rows) {
    Console.WriteLine(row[0]+" "+row[1]);
}

//read parent from child
DataRow rowp = dt2.Rows[2].GetParentRow(dataRelation);
Console.WriteLine(rowp[0] + " " + rowp[1]);
//if many-to-many use getParentRows
```

dt2 ⇒ ScoreTable

scoreId	score	studentId

dt ⇒ StudentTable

studentId	studentName

Fill()

adapter.Fill();

- It is used to add rows in the DataSet to match those in the data source.
- no matter the status of connection, after this adapter.Fill(), the connection status will stay the same as before executing this.
- During adapter.Fill(), adapter is the one to control the connection
- Retrieve data and sink to memory at once

Manually - faster:

```
conn.Open()
adapter.Fill()
conn.Close()
```

Automatically:

```
adapter.Fill()
```

DataSet&DataAdapter

- The table names will be a problem while using Fill() method .

- By default, the DataSet will assign table names as “Table, Table1 ...Table n”.

```
string query = "SELECT * FROM StudentClass; SELECT * FROM StudentInfo;";
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    DataSet dataSet = new DataSet();
    SqlDataAdapter adapter = new SqlDataAdapter(query, conn);
    //Now 2 tables are populated in the data set, the default names are "Table" and "Table1"
    //adapter.Fill(dataSet);
    //Name the tables
    //dataSet.Tables[0].TableName = "StudentClass";
    //dataSet.Tables[1].TableName = "StudentInfo";
    //First Table
    Console.WriteLine("Table 1 Data");
    foreach (DataRow row in dataSet.Tables["Table"].Rows)
    {
        Console.WriteLine(row["ClassId"] + ",  " + row["ClassName"]);
    }
    Console.WriteLine();

    // Second Table
    Console.WriteLine("Table 2 Data");
    foreach (DataRow row in dataSet.Tables["Table1"].Rows)
    {
        Console.WriteLine(row["StuId"] + ",  " + row["StuName"] );
    }
}
```

How to reflect changes from dataset to database?

Update() with CommandBuilder (Automatically)

adapter.Update()

- The Update() method belongs to the DataAdapter
- It is used to call the respective INSERT, UPDATE or DELETE statement
- It passes in a DataSet or a DataTable, and returns an int – infected row number

CommandBuilder

A CommandBuilder object is used to create INSERT, UPDATE or DELETE statement for us. Each data provider has a command builder class. We will use SqlCommandBuilder at this time.

Update() - Manually

Four Important Properties

- SelectCommand
- InsertCommand
- UpdateCommand
- DeleteCommand

Steps:

- Create a SqlDataAdapter object accompanying the query string and connection object.
- Use the Update command of SqlDataAdapter object to execute the update query.

Update - Manually

```
SqlDataAdapter adapter = new SqlDataAdapter();
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

// Create the commands.
adapter.SelectCommand = new SqlCommand(
    "SELECT CustomerID, CompanyName FROM CUSTOMERS", connection);
adapter.InsertCommand = new SqlCommand(
    "INSERT INTO Customers (CustomerID, CompanyName) " +
    "VALUES (@CustomerID, @CompanyName)", connection);
adapter.UpdateCommand = new SqlCommand(
    "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
    "WHERE CustomerID = @oldCustomerID", connection);
adapter.DeleteCommand = new SqlCommand(
    "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);
```

```
// Create the parameters.
adapter.InsertCommand.Parameters.Add("@CustomerID",
    SqlDbType.Char, 5, "CustomerID");
adapter.InsertCommand.Parameters.Add("@CompanyName",
    SqlDbType.VarChar, 40, "CompanyName");
                        Match to the Column name
adapter.UpdateCommand.Parameters.Add("@CustomerID",
    SqlDbType.Char, 5, "CustomerID");
adapter.UpdateCommand.Parameters.Add("@CompanyName",
    SqlDbType.VarChar, 40, "CompanyName");
adapter.UpdateCommand.Parameters.Add("@oldCustomerID",
    SqlDbType.Char, 5, "CustomerID").SourceVersion =
    DataRowVersion.Original;

adapter.DeleteCommand.Parameters.Add("@CustomerID",
    SqlDbType.Char, 5, "CustomerID").SourceVersion =
    DataRowVersion.Original;
```

DataAdapter

Populating a DataSet from Multiple DataAdapters

```
// Assumes that customerConnection is a valid SqlConnection object.
// Assumes that orderConnection is a valid OleDbConnection object.
SqlDataAdapter custAdapter = new SqlDataAdapter(
  "SELECT * FROM dbo.Customers", customerConnection);
OleDbDataAdapter ordAdapter = new OleDbDataAdapter(
  "SELECT * FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();

custAdapter.Fill(customerOrders, "Customers");
ordAdapter.Fill(customerOrders, "Orders");

DataRelation relation = customerOrders.Relations.Add("CustOrders",
  customerOrders.Tables["Customers"].Columns["CustomerID"],
  customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)
{
  Console.WriteLine(pRow["CustomerID"]);
   foreach (DataRow cRow in pRow.GetChildRows(relation))
    Console.WriteLine("\t" + cRow["OrderID"]);
}
```

DataAdapter

- Execute StoredProcedure in Sql Server

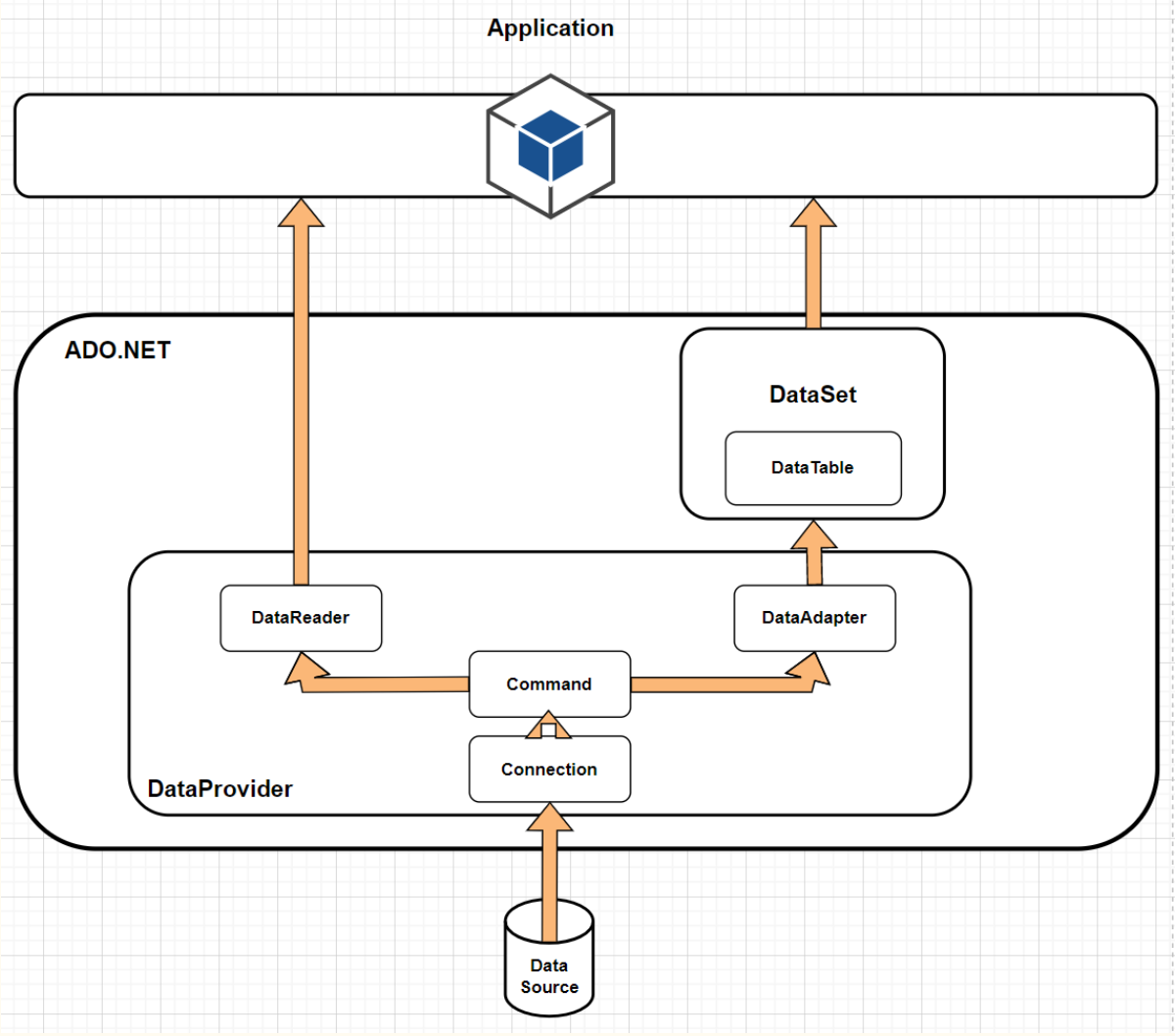
```sql
CREATE PROCEDURE GetAllClassName AS
BEGIN
    SELECT ClassName FROM StudentClass
END

GO
```

```csharp
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    Console.WriteLine("-------------------------");
    Console.WriteLine("DataTable Result(Stored Procedure) : ");
    SqlDataAdapter adapter = new SqlDataAdapter("GetAllClassName", conn);
    //Specify the Command type as Stored Procedure
    adapter.SelectCommand.CommandType = CommandType.StoredProcedure;

    DataTable dataTable = new DataTable();
    adapter.Fill(dataTable);
    foreach (DataRow row in dataTable.Rows)
    {
        Console.WriteLine(row["ClassName"]);
    }
}
```

Architecture Recap

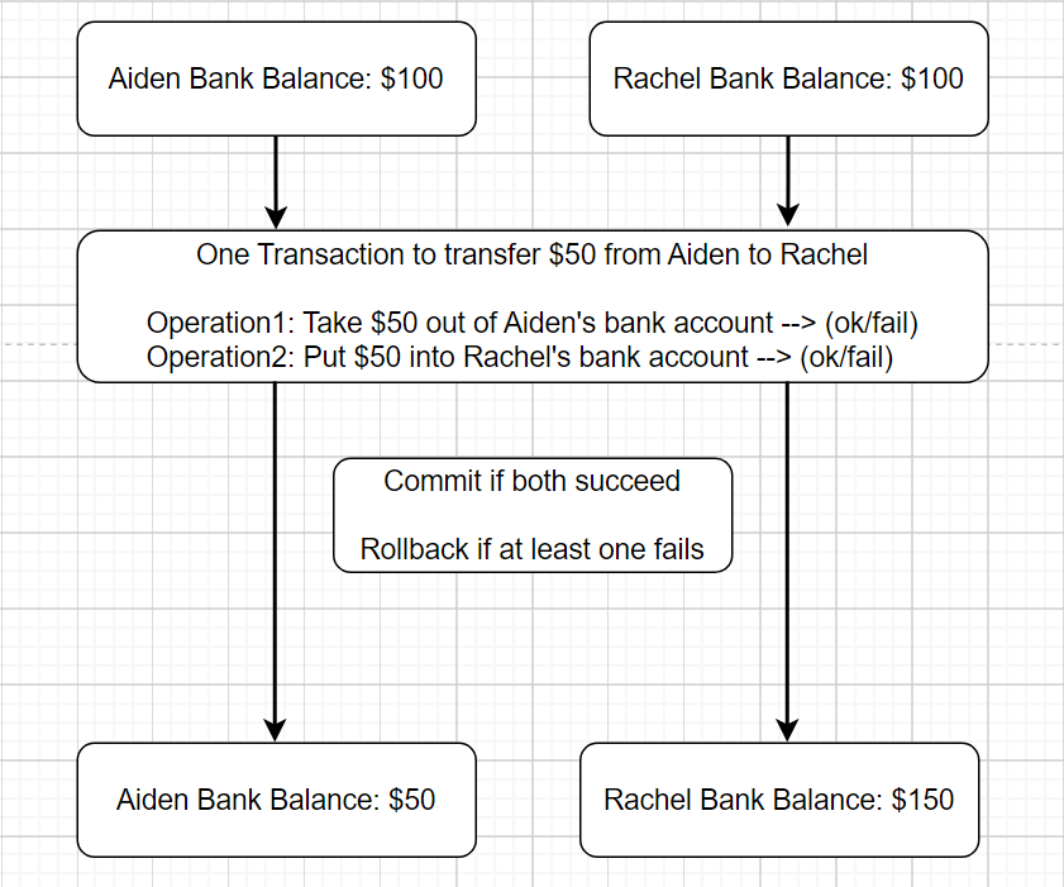


Adapter VS. Reader

	DataReader	**DataAdapter**
Speed	fast	slow
DataVolume	small	large
Memory Cost	One row at a time	All results
Connection status	Always occupied	Connect and disconnect
Read	Read only	Can Update/Delete/Insert

Transaction Management in ADO.NET

Transaction



Transaction in ADO.NET

Transaction represents a single unit of work.

The ACID properties describes the transaction management well.

- **A**tomicity: all queries in a transaction must succeed. If one fails, all should rollback.
- **C**onsistency: the database must be consistent before and after the transaction.
- **I**solation: multiple Transactions occur independently without interference.
- **D**urability: committed transaction must be persisted in a durable storage(database).

Transaction in ADO.NET

Transactions in ADO.NET are used when you want to bind multiple tasks together so that they execute as a single unit of work.

Transaction control is performed by the **Connection** object,

You can initiate a local transaction with the **BeginTransaction** method.

Once you have begun a transaction, you can enlist a command in that transaction with the **Transaction** property of a **Command** object.

You can then **commit** or **roll back** modifications made at the data source based on the success or failure of the components of the transaction.

Transaction in ADO.NET

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();

    // Start a local transaction.
    SqlTransaction sqlTran = connection.BeginTransaction();

    // Enlist a command in the current transaction.
    SqlCommand command = connection.CreateCommand();
    command.Transaction = sqlTran;

    try
    {
        // Execute two separate commands.
        command.CommandText =
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')";
        command.ExecuteNonQuery();
        command.CommandText =
          "INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')";
        command.ExecuteNonQuery();

        // Commit the transaction.
        sqlTran.Commit();
        Console.WriteLine("Both records were written to database.");
    }
    catch (Exception ex)
    {
        // Handle the exception if the transaction fails to commit.
        Console.WriteLine(ex.Message);

        try
        {
            // Attempt to roll back the transaction.
            sqlTran.Rollback();
        }
        catch (Exception exRollback)
        {
            // Throws an InvalidOperationException if the connection
            // is closed or the transaction has already been rolled
            // back on the server.
            Console.WriteLine(exRollback.Message);
        }
    }
}
```

Exception Handling in ADO.NET

- Programs should recover and leave the database in a consistent state.
- If a statement in the try block throws an exception or warning, it can be caught in one of the corresponding catch statements
- How might a finally {...} block be helpful here?
    - E.g., you could rollback your transaction in a catch { ...} block or close database connection and free database related resources in finally {...} block

Questions?